(2)

MASSACHUSETTS INSTITUTE OF TECHNOLOGY VLSI PUBLICATIONS

AD-A211 909

VLSI Memo No. 89-532
May 1989

The J-Machine: A Fine-Grain Concurrent Computer

William J. Dally, Andrew Chien, Stuart Fiske, Waldemar Horwat, John Keen, Michael Larivee, Rich Lethin, Peter Nuth, Scott Wills, Paul Carrick, and Greg Fyler

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

Abstract

The J-Machine is a fine-grain concurrent computer that provides low-overhead primitive mechanisms for communication, synchronization, and translation. Communication mechanisms are provided that permit a node to send a message to any other node in the machine in < 2μs. On message arrival, a task is created and dispatched in < 1μs. A translation mechanism supports a global virtual address space. These mechanisms efficiently support most proposed models of concurrent computation. The hardware is an ensemble of up to 65,536 nodes each containing a 36-bit processor, 4K 36-bit words of memory, and a router. The nodes are connected by a high-speed 3-D mesh network. This design was chosen to make the most efficient use of available chip and board area.

DTIC
ELECTE
SEP 0 5 1989
S D
D

89 9 01 035

Microsystems Research Center Room 39-321
Massachusetts Institute of Technology
Cambridge Massachusetts 02139
Telephone (617) 253-8138

Acknowledgements

To appear in the *IFIP Congress 89*, San Francisco, CA, Aug./Sept. 1989. This research was supported in part by the Defense Advanced Research Projects Agency under contracts N00014-88-K-0738, N00014-87-K-0825, and N00014-85-K-0124, by a National Science Foundation Presidential Young Investigator Award with matching funds from General Electric Corporation and International Business Machines Corporation, and in part by Intel Corporation.

Author Information

Dally: Artificial Intelligence Laboratory and the Laboratory for Computer Science, Room NE43-417, MIT, Cambridge, MA 02139. (617) 253-6043.
Keen, Larivee, Lethin, and Fiske: Artificial Intelligence Laboratory and the Laboratory for Computer Science, Room NE43-416, MIT, Cambridge, MA 02139. (617) 253-8473.
Chien, Horwat, Nuth, and Wills: Artificial Intelligence Laboratory and the Laboratory for Computer Science, Room NE43-415, MIT, Cambridge, MA 02139. (617) 253-6048.
Carrick & Fyler: Intel Corporation, Santa Clara, CA 97006.

Copyright© 1989 MIT. Memos in this series are for use inside MIT and are not considered to be published merely by virtue of appearing in this series. This copy is for private circulation only and may not be further copied or distributed, except for government purposes, if the paper acknowledges U. S. Government sponsorship. References to this work should be either to the published version, if any, or in the form "private communication." For information about the ideas expressed herein, contact the author directly. For information about this series, contact Microsystems Research Center, Room 39-321, MIT, Cambridge, MA 02139; (617) 253-8138.

The J-Machine: A Fine-Grain Concurrent Computer [1]

William J. Dally, Andrew Chien, Stuart Fiske, Waldemar Horwat, John Keen, Michael Larivee, Rich Lethin, Peter Nuth, Scott Wills
Artificial Intelligence Laboratory
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Paul Carrick, Greg Fyler Intel Corporation
Santa Clara, CA 97006

Abstract

The J-Machine is a fine-grain concurrent computer that provides low-overhead primitive mechanisms for communication, synchronization, and translation. Communication mechanisms are provided that permit a node to send a message to any other node in the machine in $< 2\mu s$. On message arrival, a task is created and dispatched in $< 1\mu s$. A translation mechanism supports a global virtual address space. These mechanisms efficiently support most proposed models of concurrent computation. The hardware is an ensemble of up to 65,536 nodes each containing a 36-bit processor, 4K 36-bit words of memory, and a router. The nodes are connected by a high-speed 3-D mesh network. This design was chosen to make the most efficient use of available chip and board area.

1 Introduction

Overview

The J-Machine is a distributed-memory, MIMD, concurrent computer [11]. It provides primitive mechanisms for communication, synchronization, and translation. Communication mechanisms are provided that permit a node to send a message to any other node in the machine in $< 2\mu s$. No processing resources on intermediate nodes are consumed and buffer memory is automatically allocated on the receiving node. The synchronization mechanisms schedule and dispatch a task in $< 1\mu s$ on message arrival and suspend tasks that attempt to reference data that is not yet available. The translation mechanism maintains bindings between arbitrary names and values. It is used to perform address translation to support a global virtual address space.

These mechanisms have been selected to be both general and amenable to efficient hardware implementation. They efficiently support many parallel models of computation including actors[1].

[1]The research described in this paper was supported in part by the Defense Advanced Research Projects Agency under contracts N00014-88K-0738, N00014-87KC-0825, and N00014-85-K-0124, in part by a National Science Foundation Presidential Young Investigator Award with matching funds from General Electric Corporation and International Business Machines Corporation, and in part by Intel Corporation.

per ltr
Codes
A-1

dataflow[16], and communicating processes[19]. To support fine-grain concurrent programming systems, the mechanisms are designed to efficiently handle small objects (8 words) and small tasks (20 instructions).

The hardware is an ensemble of up to 65,536 message-driven processors (MDPs)[9]. This limit is set by the addressability of the router and the bandwidth of the network. Each node contains a 36-bit processor, 4K 36-bit words of memory, and a communications controller (router). The nodes are connected by a high-speed, three-dimensional mesh network. Each network channel has a bandwidth of 450Mbits/s. The first medium-scale prototype will be a 4096-node system.

This design was chosen to make the most efficient use of available chip and board area. Packaging a small amount of memory on each node gives us an extremely high memory bandwidth (3Gbits/s per chip or 200Tbits/s in a fully populated system). Memory consumes most of the chip area; from one point of view, the system is a memory with processors added to each node to improve bandwidth for local operations. The fast communication and global address space prevent the small local memories from limiting programmability or performance. The 3-D network gives the highest throughput and lowest latency for a given wire density[7][13]. It allows the processing nodes to be packed densely and results in uniformly short wires.

The J-Machine project is driven by the following goals:

- To identify and implement simple hardware mechanisms for communication, synchronization, and naming suitable for supporting a broad range of concurrent programming models.
- To reduce the overhead associated with these mechanisms to a few instruction times so that fine-grain programs may be efficiently executed.
- To design an area-efficient machine: one that maximizes performance for a given amount of chip and wiring area.

Grain Size

The J-Machine is a fine-grain concurrent computer in that (1) [illegible] designed to efficiently support fine-grain programs and (2) it is composed of fine-grain processing nodes [14].

The *grain size* of a program refers to the size of the tasks and messages that make up the program. Coarse-grain programs have a few long ($\approx 10^5$ instruction) tasks, while fine-grain programs have many short (≈ 20 instruction) tasks. With more tasks that can execute at a given time – viz. more concurrency – fine-grain programs (in the absence of overhead) result in faster solutions than coarse-grain programs.

The *grain size* of a machine refers to the physical size and the amount of memory in one processing node. A coarse-grain processing node requires hundreds of chips (several boards) and has $\approx 10^7$ bytes of memory while fine-grain node fits on a single chip and has $\approx 10^4$ bytes of memory. Fine-grain nodes cost less and have less memory than coarse-grain nodes, however.

because so little silicon area is required to build a fast processor, they need not have slower processors than coarse-grain nodes.

Background

The J-Machine builds on previous work in the design of message-passing and shared memory machines. Like the Caltech Cosmic Cube [25], the Intel iPSC [21], the N-CUBE [23], and the Ametek [2], each node of the J-Machine has a local memory and communicates with other nodes by passing messages. Because of its low overhead, the J-Machine can exploit concurrency at a much finer grain than these early message passing computers. Delivering a message and dispatching a task in response to the message arrival takes $< 3\mu s$ on the J-Machine as opposed to 5ms on an iPSC. Like the BBN Butterfly [4] and the IBM RP3 [24] the J-Machine provides a global virtual address space. The same IDs (virtual addresses) are used to reference on and off node objects. Like the InMOS transputer [20] and the Caltech MOSAIC [22] a J-Machine node is a single chip processing element integrating a processor, memory, and a communication unit.

The J-Machine is unique in that it extends these previous efforts with efficient primitive mechanisms for communication, synchronization and naming.

Summary

The remainder of this paper describes the J-Machine in more detail Section 2 gives an overview of the system describing how the network and processor work together to support concurrent programming systems. The network is described in Section 3. Section 4 deals with the message driven processor and the mechanisms it provides for concurrency.

2 System Architecture

The J-Machine consists of an ensemble of (up to 64K) MDP-based processing nodes connected by a three-dimensional mesh network. The network delivers messages between nodes. All routing and flow-control are handled in the network. The network consists of a communication controller or router on each node [6][10][17], and wires connecting these routers to their neighbors in each of the three physical dimensions.

Each node contains a memory, a processor, and a communication controller. The communication controller is logically part of the network but physically part of the node. The 4K-word by 36-bit memory is used to store objects and system tables. Each word of the memory contains a 32-bit data item and a 4-bit tag. In addition to the usual uses of tags to support dynamic typing and garbage collection, special tags are provided to synchronize on data presence and to indicate if an address is local or remote. Memory accesses to write messages or read code are made a row (144bits) at a time to improve memory bandwidth. A part of the memory can be

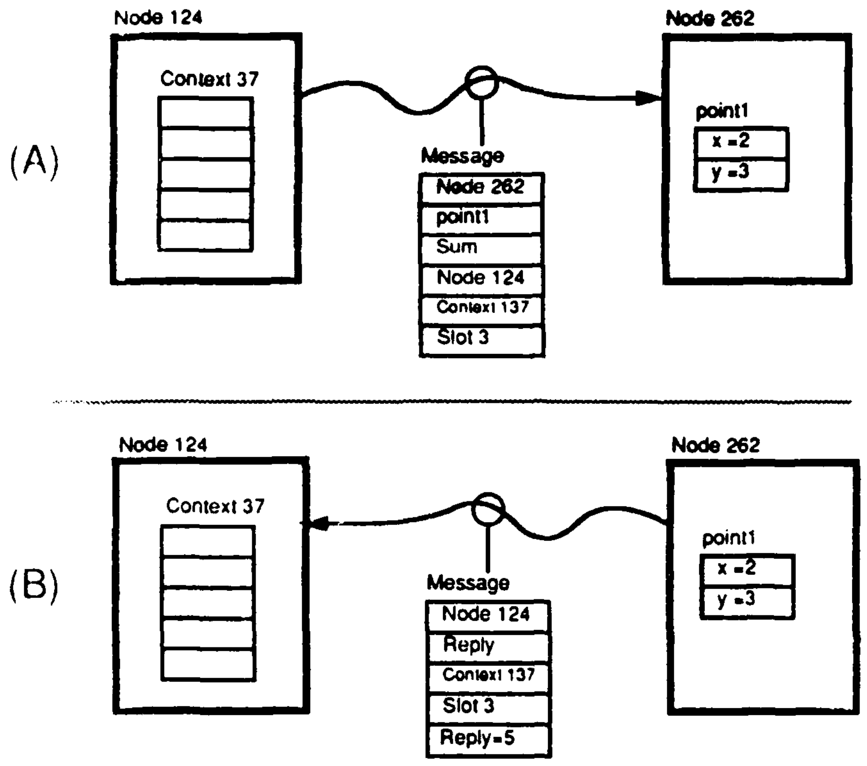

Figure 1: (a) A task executing in Context 37 on Node 124 sends a message to object point1 requesting that it perform the Sum method. (b) A reply message is returned to Context 37.

mapped as a set associative cache. This cache is used to implement the processor's translation mechanism.

The processor is message driven. It executes user and system code in response to messages arriving over the network. A conventional processor is instruction-driven in that it fetches an instruction from memory and dispatches a control sequence to execute the instruction. A message-driven processor receives a message from the network and dispatches a control sequence to execute the task specified by the message. The MDP uses an instruction sequence to *execute* a message. Hardware mechanisms for communication, synchronization, and translation are provided to accelerate the dispatch operation and the subsequent task execution.

To support communication over the network, the MDP provides a **SEND** instruction and performs automatic buffering of arriving messages. To synchronize execution with arriving messages, a primitive dispatch operation is provided that eliminates scheduling overhead. To synchronize on data, tags are provided to support futures. A general translation mechanism uses a set associative cache in the node memory to maintain arbitrary bindings.

To see how the system functions together, consider the example shown in Figure 1. In Figure 1a, A task executing in Context 37 on Node 124 sends a Sum message to an object, point1. This message requests that the object sum its two fields x and y. The sending node translates the object name for point1 (a unique 32-bit pattern) into a node address, Node 262 (a 16-bit integer), using the MDP translation mechanism. A sequence of MDP SEND instructions is then used to inject the message into the network. The message includes (1) the node address of point1 (Node 262), (2) the object name of point1, (3) the message name or selector, Sum, and (4) a continuation (the node and ID of the sender's context and the slot into which the reply should be stored). The sending task continues to execute until it needs the result of the Sum message.

The network delivers the injected message to Node 262. At this node, the MDP buffering mechanism allocates storage for the message and sequences the message off the network into the node's memory. When the node completes its current task (and any other tasks ahead in the queue), the MDP dispatch mechanism creates a new task in response to the message. This task translates the ID of point1 into a segment descriptor for the object, adds the x and y fields of the object together, and uses a sequence of SEND instructions to inject a message containing the sum into the network. As shown in Figure 1b, this message contains (1) the node address of the sender's context, (2) the ID of the sender's context, (3) the context slot awaiting the reply, and (4) the result. This task then terminates.

The network delivers the reply message to Node 124 where it is buffered and eventually dispatched to create a task. This task translates the ID for Context 37 into a segment descriptor. The reply value is stored into the specified slot of this context. The sending task is then resumed by loading its context from this segment.

The round trip delay for this example message send and reply is $\approx 5\mu s$. The difficulty in building a concurrent system the scale of the J-machine is not developing the mechanisms conceptually. It is implementing them efficiently so the overhead of accessing remote nodes is made small enough to permit the execution of fine-grain programs.

In the following sections we will examine the implementation of each component of the J-Machine system.

3 The Network

The J-Machine network has a 3-D mesh topology as shown in Figure 2. Each node is located by a three coordinate address (x, y, and z). A node is connected to its six neighbors (if they exist) that have addresses differing in only one coordinate by ± 1. All connections are bidirectional channels. Each channel requires 15 wires to carry 9-data bits, one tail bit, and five control lines [10]. Addressing is provided to support up to a $32 \times 32 \times 64$ cube of 65536 nodes. The prototype will be built as a $16 \times 16 \times 16$ cube of 4096 nodes. For a machine, such as the J-Machine where wire density is a limiting factor, this topology has been shown to give the lowest latency and highest throughput for a given wire density [7][13].

The network topology is not visible to the programmer. The latency of sending a message from

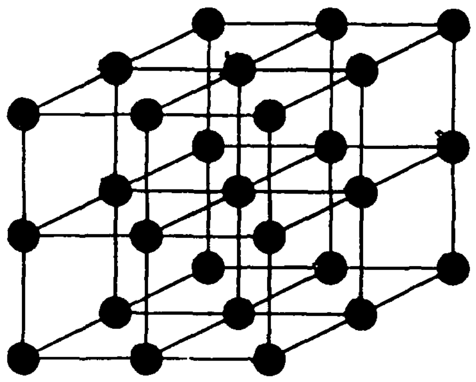
Figure 2: The J-Machine Network is a 3-D mesh or k-ary 3-cube (a $3\times 3\times 3$ mesh is shown here). Messages injected into the network at any node are routed to the destination node specified in the head of the message. All routing and flow control is performed in the network.

any node, i, to any other node, j, is sufficiently low that the programmer sees the network as a complete connection. Zero load network latency is given by

$$T = T_d D + T_c \frac{L}{W}. \tag{1}$$

Where D is the distance (number of hops) the message must travel, L is the length of the message in bits, and W is the width of the channel in bits. The network is expected to have a propagation delay per stage, T_d, of 20ns and a channel cycle time, T_c of 20ns. With these times, a six word (L =216 bit) message traversing half the network diameter ($D = 24$) has a latency of 960ns equally divided between the two components of latency [13]. An average message travels one third of the network diameter for a latency of 800ns.

The network provides all end to end message delivery services. The sending node injects a message containing the absolute address of the destination node. The network determines the route of the message, and sequences each flit (flow-control digit) of the message over the route. Flow control is performed as required to resolve contention and match channel rates. This flow control is performed in a manner that is provably deadlock free[8].

There is no acknowledgement, error detection, or error correction on the network channels. The network wires are all short, contained within a single physical cabinet, and operated at low impedance. The error rate of a network channel is no higher than that of a properly terminated signal in a conventional CPU.

4 The Message-Driven Processor

The message-driven processor (MDP) is a 36 bit single-chip microcomputer specialized to oper ate efficiently in a multicomputer. [5][9][12]. The MDP chip includes the processor, a 4K word

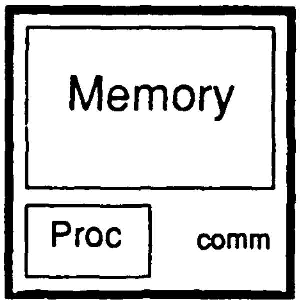

Figure 3: The Message-Driven Processor chip incorporates a 36-bit processor, a 4K-word × 36-bit memory, and a router (described above).

by 36-bit memory, and a router (Figure 3). An on-chip memory controller with ECC permits local memory to be expanded up to 1M-words by adding external DRAM chips.

Other machines have combined processor, memory, and communications on a single chip [20] [22] [23]. The MDP extends this work by providing fast, primitive mechanisms for synchronization, communication, and translation (naming) that allow the processor to efficiently support many parallel models of computation. A fast network is of little use if very large overheads are required to initiate and receive messages at the processing nodes. The MDP's mechanisms reduce the overhead of interacting with other processors over the network to levels that make fine-grain parallelism efficient.

The following mechanisms are provided:

- **Communication Mechanisms**
  - A `SEND` instruction injects messages into the network.
  - Messages arriving from the network are automatically buffered in a circular queue.
- **Synchronization Mechanisms**
  - A dispatch mechanism creates and schedules a task (thread of control and addressing environment) to handle each arriving message.
  - Tags for *futures* [3] synchronize tasks based on data dependencies.
- **Translation**
  - `ENTER` and `XLATE` (translate) instructions make bindings between arbitrary 36-bit key and data values (`ENTER`) and retrieve a value given the corresponding key (`XLATE`).
  - Segmented memory management provides relocation and protection for data objects stored in a node's memory.

```
SEND     R0          ; send net address
SEND2    R1,R2       ; header and receiver
SEND2E   R3,[3,A3]   ; selector and continuation - end msg.
```

Figure 4: MDP assembly code to send a 4 word message uses three variants of the SEND instruction.

The processor is *message driven* in the sense that processing is performed in response to messages (via the dispatch mechanism). There is no receive instruction. A task is created for each arriving message to handle that message. A computation is advanced (driven) by the messages carrying tasks about the network.

4.1 Send Instruction

The MDP injects messages into the network using a send instruction that transmits one or two words (at most one from memory) and optionally terminates the message. The first word of the message is interpreted by the network as an absolute node address (in x,y format) and is stripped off before delivery. The remainder of the message is transmitted without modification. A typical message send is shown in Figure 4. The first instruction sends the absolute address of the destination node (contained in R0). The second instruction sends two words of data (from R1 and R2). The final instruction sends two additional words of data, one from R3, and one from memory. The use of the SENDE instruction marks the end of the message and causes it to be transmitted into the network. In a Concurrent Smalltalk message [15], the first word is a message header, the second specifies the receiver, the third word is the selector, subsequent words contain arguments, and the final word is a continuation. This sequence executes in 4 clock cycles (200ns).

A first-in-first-out (FIFO) buffer is used to match the speed of message transmission to the network. In some cases, the MDP cannot send message words as fast as the network can transmit them. Without a buffer, *bubbles* (absence of words) would be injected into the network pipeline degrading performance. The SEND instruction loads one or two words into the buffer. When the message is complete or the eight-word buffer is full, the contents of the buffer are launched into the network.

Previous concurrent computers have used direct-memory access (DMA) or I/O channels to inject messages into the network. First an instruction sequence composed a message in memory. DMA registers or channel command words were then set up to initiate sending. Finally, the DMA controller transferred the words from the memory into the network. This approach to message sending is too slow for two reasons. First, the entire message must be transferred across the memory interface twice, once to compose it in memory and a second time to transfer it into the network. Second, for very short messages, the time required to set up the DMA control registers or I/O channel command words often exceeds the time to simply send the message into the network.

4.2 Message Reception

The MDP maintains two message/scheduling queues (one for each priority level) in its on-chip memory. The queues are implemented as circular buffers. As messages arrive over the network, they are buffered in the appropriate queue. To improve memory bandwidth, messages are enqueued by rows. Incoming message words are accumulated in a row buffer until the row buffer is filled or the message is complete. The row buffer is then written to memory.

It is important that the queue have sufficient performance to accept words from the network at the same rate at which they arrive. Otherwise, messages would backup into the network causing congestion. The queue row buffers in combination with hardware update of queue pointers allow enqueuing to proceed using one memory cycle for each four words received. Thus a program can execute in parallel with message reception with little loss of memory bandwidth.

Providing hardware support for allocation of memory in a circular buffer on a multicomputer is analogous to the support provided for allocation of memory in push-down stacks on a uniprocessor. Each message stored in the MDP message queue represents a method activation much as each stack frame allocated on a push-down stack represents a procedure activation.

4.3 Dispatch

Each message in the queues of an MDP represents a task that is ready to run. When the message reaches the head of the queue, a task is created to handle the message. At any time, the MDP is executing the task associated with the first message in the highest priority non-empty queue. If both queues are empty, the MDP is idle - viz., executing a background task. Sending a message implicitly schedules a task on the destination node. This simple two-priority scheduling mechanism removes the overhead associated with a software scheduler. More sophisticated scheduling policies may be implemented on top of this substrate.

Messages become *active* either by arriving while the node is idle or executing at a lower priority, or by being at the head of a queue when the preceding message *suspends* execution. When a message becomes active a task is created to handle it. Task creation, changing the thread of control and creating a new addressing environment, are performed in one clock cycle as shown in Figure 5. Every message header contains a message *opcode* and the message *length*. The message opcode is loaded into the IP to start a new thread of control. The length field is used along with the queue head to create a message segment descriptor that represents the initial addressing environment for the task. The message handler code may open additional segments by translating object IDs in the message into segment descriptors.

No state is saved when a task is created. If a task is preempting lower priority execution, it executes in a separate set of registers. If a task, A, becomes active when an earlier task, B, at the same priority suspends, B is responsible for saving its live state before suspending.

The dispatch mechanism is used directly to process messages requiring low latency (e.g., combining and forwarding). Other messages (e.g., remote procedure call) specify a handler that locates the required method (using the translation mechanism described below) and then trans-

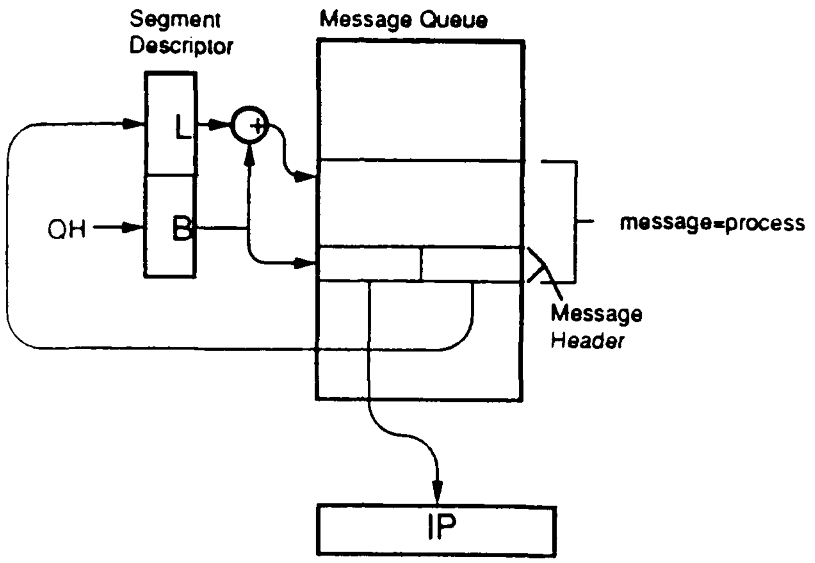

Figure 5: Message dispatch. In one clock cycle, a new task is created by (1) setting the IP to change the thread of control and (2) creating a message segment to provide the initial addressing environment.

```
MOVE    [1,A3],R0    ; get method id
XLATE   R0,A0        ; translate to segment descriptor
LDIP    INITIAL_IP   ; transfer control to method
```

Figure 6: MDP assembly code for the CALL message.

fers control to it.

For example, a remote procedure call message is handled by the call handler code as shown in Figure 6. The execution of this handler is depicted in Figure 7. The first instruction gets the method ID (offset 1 into the message segment reference by A3). The next instruction translates this method ID into a segment descriptor for the method and places this descriptor in A0. The final instruction transfers control to the method. The method code may then read in arguments from the message queue. The argument object identifiers are translated to physical memory base/length pairs using the translate instruction. If the method needs space to store local state, it may create a context object. When the method has finished execution, or when it needs to wait for a reply, it executes a SUSPEND instruction passing control to the next message.

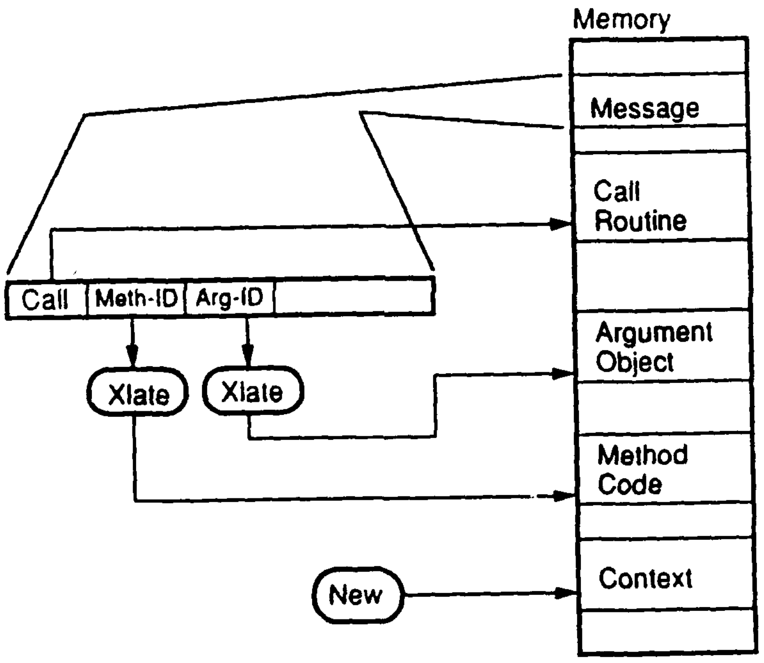

Figure 7: The CALL message invokes a method by translating the method identifier to find the code, creating a context (if necessary) to hold local state, and translating argument identifiers to locate arguments.

4.4 Synchronization with Tags

Every register and memory location in the MDP includes a 4-bit tag that indicates the type of data occupying the location. The MDP uses tags for synchronization on data availability in addition to their conventional uses for dynamic typing and run-time type checking. Two tags are provided for synchronization: **future**, and **c-future**. A **future** tag is used to identify a named placeholder for data that is not yet available [3]. Applying a strict operator to a **future** causes a fault. A **future** can, however, be copied without faulting. A **c-future** tag identifies a cell awaiting data. Applying any operator to a **c-future** causes a fault. As they are unnamed placeholders, they cannot be copied.

The **c-future** tag is used to suspend a task if it attempts to access data that has not yet arrived from a remote node. When a task sends a message requesting a reply, it marks the cell that will hold the reply as a **c-future**. Any attempt to reference the reply before it is available will fault and suspend the task. When the reply arrives, it overwrites the **c-future** and resumes the task if it was suspended. For example, when the task executing in Context 37 in Figure 1 sends the **Sum** message, it marks Slot 3 of its context as a **c-future**. The reply message overwrites Slot 3 to indicate data presence.

The **future** tag is used to implement named futures as in Multilisp [18]. **Futures** are more general than **c-futures** in that they can be copied. However, they are much more expensive than **c-futures**. A memory area and a name must be allocated for each future generated.

4.5 Translation

The MDP is an experiment in unifying shared-memory and message-passing parallel computers. Shared-memory machines provide a uniform global name space (address space) that allows processing elements to access data regardless of its location. Message-passing machines perform communication and synchronization via node-to-node messages. These two concepts are not mutually exclusive. The MDP provides a virtual addressing mechanism intended to support a global name space while using an execution mechanism based on message passing.

The MDP implements a global virtual address space using a general translation mechanism. The MDP memory allows both indexed and set-associative access. By building comparators into the column multiplexer of the on-chip RAM, we are able to provide set-associative access with only a small increase in the size of the RAM's peripheral circuitry.

The translation mechanism is exposed to the programmer with the **ENTER** and **XLATE** instructions. **ENTER Ra,Rb** associates the contents of **Ra** (the key) with the contents of **Rb** (the data). The association is made on the full 36 bits of the key so that tags may be used to distinguish different keys. **XLATE Ra,Rb** looks up the data associated with the contents of **Ra** and stores this data in **Rb**. The instruction faults if the lookup *misses*. This mechanism is used by our system code to cache ID to segment descriptor (virtual to physical) translations, to cache ID to node number (virtual to physical) translations, and to cache class/selector to segment descriptor (method lookup) translations.

Tags are an integral part of our addressing mechanism. An ID may translate into a segment descriptor for a local object, or a node address for a global object. The tag allows us to distinguish these two cases and a fault provides an efficient mechanism for the test. Tags also allow us to distinguish an ID key from a class/selector key with the same bit pattern.

Most computers provide a set associative cache to accelerate translations. We have taken this mechanism and exposed it in a pair of instructions that a systems programmer can use for any translation. Providing this general mechanism gives us the freedom to experiment with different address translation mechanisms and different uses of translation. We pay very little for this flexibility since performance is limited by the number of memory accesses that must be performed.

5 Conclusion

The J-Machine is a general purpose parallel computer. It provides general mechanisms for communication, synchronization, and translation rather than hardwiring mechanisms for a specific model of computation. These mechanisms efficiently support many proposed models of computation. Using these mechanisms, the overhead of creating a task on a remote node is reduced to a few microseconds. This low overhead permits concurrency to be exploited at a fine-grain size.

The J-Machine is designed to make efficient use of silicon and wiring area. Each message driven processing node is a *jellybean* part. It can be fabricated in the same technology used to manufacture existing commodity semiconductor parts such as DRAMs. The network is designed to make efficient use of wires so the machine can be packaged densely – with processing nodes consuming most of the volume. There are no large wiring channels.

At the time of this writing (January 1988), the project is currently in the advanced design stage. Message-level, instruction-level, and register-transfer-level simulators have been built to test the J-Machine design. Prototype versions of JOSS and the CST compiler are operational. Gate and transistor level schematics are in the process of being drawn. We expect to complete the processing node chip design in late 1989 and have a prototype J-Machine System running in mid 1990.

Acknowledgement

The following MIT students have contributed to the work described here: Linda Chao, Soha Hassoun. Paul Song, and Brian Totty.

I thank Tom Knight, Gerry Sussman, Steve Ward, Dave Gifford, Tom Leighton, and Carl Hewitt of MIT. Chuck Seitz and Bill Athas of Caltech. and Mark Vestrich, Albert Yu, Justin Rattner, and George Cox of Intel Corporation for many valuable suggestions comments, and advice.

References

[1] Agha, Gul A., *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, MA, 1986.

[2] Ametek Computer Research Division, *Series 2010 Product Description*, 1987.

[3] Baker, H. and Hewitt, C., "The Incremental Garbage Collection of Processes," *ACM Conference on AI and Programming Languages*, Rochester, New York, August, 1977, pp. 55-59.

[4] BBN Advanced Computers, Inc., *Butterfly Parallel Processor Overview*, BBN Report No. 6148, March 1986.

[5] Dally, William J., *A VLSI Architecture for Concurrent Data Structures*, Kluwer, Hingham, MA, 1987.

[6] Dally, William J. and Seitz, Charles L., "The Torus Routing Chip," *J. Distributed Systems*, Vol. 1, No. 3, 1986, pp. 187-196.

[7] Dally, William J. "Wire Efficient VLSI Multiprocessor Communication Networks," *Proceedings Stanford Conference on Advanced Research in VLSI*, Paul Losleben, Ed., MIT Press, Cambridge, MA, March 1987, pp. 391-415.

[8] Dally, William J. and Seitz, Charles L., " Deadlock-Free Message Routing in Multiprocessor Interconnection Networks," *IEEE Transactions on Computers*, Vol. C-36, No. 5, May 1987, pp. 547-553.

[9] Dally, William J. et.al., "Architecture of a Message-Driven Processor," *Proceedings of the 14th ACM/IEEE Symposium on Computer Architecture*, June 1987, pp. 189-196..

[10] Dally, William J., and Song, Paul., "Design of a Self-Timed VLSI Multicomputer Communication Controller," *Proc. International Conference on Computer Design, ICCD-87*, 1987, pp. 230-234.

[11] Dally, William J., "The J-Machine: System Support for Actors," *Actors: Knowledge-Based Concurrent Computing*, Hewitt and Agha eds., MIT Press, 1989.

[12] Dally, William J. et.al., *Message-Driven Processor Architecture, Version 11* MIT Artificial Intelligence Laboratory Memo No. 1069, August, 1988.

[13] Dally, William J. "Performance Analysis of k-ary n-cube Interconnection Networks," *IEEE Transactions on Computers*, to appear.

[14] Dally, W.J., "Fine-Grain Concurrent Computers", *Proc.* 3rd *Symposium on Hypercube Concurrent Computers and Applications*, ACM 1988.

[15] Dally, W.J., and Chien A.A., "Object Oriented Concurrent Programming in CST," *Proc.* 3rd *Symposium on Hypercube Concurrent Computers and Applications*, ACM 1988.

[16] Dennis, Jack B., "Data Flow Supercomputers," *IEEE Computer*, Vol. 13, No. 11, Nov. 1980, pp. 48-56.

[17] Flaig, Charles, M., *VLSI Mesh Routing Systems*, Technical Report 5241:TR:87, Dept. of Computer Science, California Institute of Technology, 1987.

[18] Halstead, Robert H., "Parallel Symbolic Computation," *IEEE Computer*, Vol. 19, No. 8, Aug 1986, pp. 35-43.

[19] Hoare, C.A R., "Communicating Sequential Processes," *Comm. ACM*, Vol. 21, No. 8, August 1978, pp. 666-677.

[20] Inmos Limited, *IMS T424 Reference Manual*, Order No. 72 TRN 006 00, Bristol, United Kingdom, November 1984.

[21] Intel Scientific Computers, *iPSC User's Guide*, Order No. 175455-001, Santa Clara, CA, Aug. 1985.

[22] Lutz, C., et. al., "Design of the Mosaic Element," *Proc. MIT Conference on Advanced Research in VLSI*, Artech Books, 1984, pp. 1-10.

[23] Palmer, John F., "The NCUBE Family of Parallel Supercomputers," *Proc. IEEE International Conference on Computer Design, ICCD-86*, 1986, p. 107.

[24] Pfister, G.F. et. al., "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture", *Proc. International Conference on Parallel Processing, ICPP*, 1985, pp. 764–771.

[25] Seitz, Charles L., "The Cosmic Cube", *Comm. ACM*, Vol. 28, No. 1, Jan. 1985, pp. 22-33.